JPL D-15198

Applying the SCR* Requirements Toolset to DS-1 Fault Protection

December 19, 1997

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

Applying the SCR* Requirements Toolset to DS-1 Fault Protection

Prepared by:

Robyn R. Lutz
Task Lead

Hui-Yin Shaw
Co-Lead

Approvals:

Tuyet-Lan Tran
Software Assurance Supervisor

John Kelly
ATPO Software Applications Program, PEM

December 19, 1997

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D-15198

Applying the SCR* Requirements Toolset to DS-1 Fault Protection

Robyn R. Lutz and Hui-Yin Shaw
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

December 19, 1997

1. Introduction

This report describes the application of the requirements toolset SCR* (Software Cost Reduction) from the Naval Research Laboratory (NRL) to components of the Deep Space-1 spacecraft at Jet Propulsion Laboratory. This research was funded by NASA's Office of Safety and Mission Assurance under UPN 323-08-5I.

The purpose of the research was to begin identifying and analyzing existing techniques that support software safety in the changing NASA environment. In particular, current needs for higher reliability, reusable software, rapid development, and innovative software architectures have focused attention on improved requirements analysis techniques. An accurate, usable requirements model provides a firm base for building safe systems. Conversely, errors in safety-related requirements contribute significantly to delays during integration and system testing (Lutz). Improvements that are often suggested include specifying requirements more precisely than English can; producing specifications that are easy to read, review, update, and distribute on the web; and using automated analysis tools that interface seamlessly with the specification.

The SCR* toolset is a candidate that meets these criteria. It is a "lightweight" formal method tool, meaning that it has a formal semantics and some useful automated analysis capabilities (Heitmeyer, Bull, Gasarch, and Labaw). However, it does not incorporate a theorem-prover as "heavyweight" formal method tools do (e.g., PVS, VDM, or HOL). SCR provides a readable, table-based specification, executable requirements model, automated analysis capabilities, and precise semantics. This formal semantics allows integration (existing or planned) with theorem provers, model checkers and test-case generators (Heitmeyer, Kirby, and Labaw). SCR thus can offer wide-ranging support for safety-critical requirements.

SCR is not a commercial tool. It is available at no cost from NRL for experimental applications. Constance Heitmeyer, the head of the Software Engineering Section, has several papers about SCR* at her website (http://www.nrl.navy.mil/ITD/5540/personnel/heitmeyer.html). Additional descriptions of SCR* and projects in which it has been used are in (Easterbrook and Callahan, Miller and Hoech).

This report is organized as follows. Section 2 of the report provides background information on the SCR* toolset and its uses to date, an overview of the toolset's features, and a discussion of its capabilities to support safety analyses.

Section 3 describes two applications of SCR to requirements and design modeling. In order to evaluate the toolset, the SCR* toolset was exercised on two critical Deep Space-1 components, the fault protection for the Xenon Feed System and the system fault protection standby mode transition logic. Section 3 describes the steps in this process, assuming that the reader has no familiarity with the SCR* toolset. The issues found, both with regard to the DS-1 requirements and with regard to SCR, are reported.

Section 4 presents a sample procedure using SCR for requirements and design modeling, tailored to the anticipated needs of a NASA development project.

Section 5 provides some concluding remarks and recommendations for future use. SCR has many of the features that are likely to assist projects in moving toward more rapid and better requirements development and maintenance: a readable, yet precise model; graphic and table-based windows; simulation of specifications; links to increasingly sophisticated automated checkers and provers; and support for checking safety invariants . These make SCR a valuable research tool and useful for verification of existing requirements. However, immediate transfer of SCR to spacecraft project developers is judged to be premature.

2. SCR Requirements Tool

SCR* (Software Cost Reduction) is a toolset developed by the Naval Research Laboratory (NRL) to support the process of requirements development, analysis, and maintenance. SCR was originally developed in the late 70's by David Parnas and others as a formal specification method, and was applied to the reengineering of a portion of the A-7E aircraft (Miller and Hoech). Since that time, the NRL has expanded and refined SCR, providing a formal semantics for the requirements modeling and a suite of automated tools that run off the formal specification. These include tools to edit, parse, and typecheck the requirements model. Other tools check dependencies among the variables and states, detect cyclical dependencies, identify missing cases and nondeterminism, and execute the requirements specifications and/or design logic in a simulator. The most recent version, as reported in (Bharadwaj and Heitmeyer) also provides a link to the model checker SPIN (via a translation of SCR into the language, Promela). Significant work has been done in linking SCR with the automated proof capabilities of PVS (Owre, Rushby, and Shankar; Archer and Heitmeyer).

Front-end interfaces between SCR and users of the system have also been developed. These tend to be domain-specific and aimed at better elicitation of actual requirements from customers. For example, an interface demonstrated by NRL's SCR group at RE'97 (RE'97) displayed the mockup of a cockpit display on the screen. Changes to monitored values on the display are fed into the SCR specifications, which then displays the effects of those changes on the controlled variables. Early exercising of requirements by customers has been identified as one way to forestall requirements misunderstandings between the developers and the end-users of a system, eliminating a common cause of project delays and overruns.

The advantage of SCR for analyzing critical requirements comes from the automated checking possible due to its formal semantics, combined with the readability of its tabular notation and windows-based toolset. In addition, invariant properties that must be true in every state or pair of adjacent states can be defined and tested during executions of the requirements simulator.

2.1. SCR Modeling

SCR is based on the widely used 4-variable model. The model provides a black-box specification that describes the externally visible behavior (i.e., physical changes) required. Loosely speaking, the physical outputs of the system are represented by "controlled variables" and the physical inputs of the system are represented by "monitored variables." An example from the SCR User's Guide (Kirby) of a monitored variable is "WeaponType"; an example of a controlled variable is "BombRelease."

The system to be developed is represented as a set of concurrently executing finite state machines. However, one of the obstacles with requirements modeling in SCR* at the present time is that almost all the published examples involve only a single finite state machine. This means that there is currently minimal guidance for delineating the boundary of the system to be modeled. Similarly, the decision as to what is "externally visible" is often difficult; e.g., we found that the controlled variable for one subsystem is often a monitored variable for another subsystem on the spacecraft. Since SCR* does not explicitly support hierarchical finite state machines, modeling decisions are sometimes complicated by tool considerations.

2.2. Requirements Specification in SCR

The requirements specification in SCR consists of a set of tables and a set of five dictionaries. The user creates and edits the various tables and dictionaries during the process of specification. Note, however, that for some purposes, partial specification is both possible and supported by SCR. For example, Easterbrook and Callahan report that they completed only as much of the specification as was needed to test certain properties of interest, since coverage and disjointedness were not a concern.

The dictionaries are the Constant Dictionary, the Variable Dictionary, the Type Dictionary, the Assertion Dictionary, and the Mode Class Dictionary. In addition, an Environmental Assertion Dictionary is supported by the tool, but was not described in the version of the User's Guide we had (January, 1997).

The Variable Dictionary and Constant Dictionary are similar to the standard data dictionary provided in most requirements documents, describing the name, type, and initial value of the data items to be used. The comments fields in the SCR dictionaries can be used to provide traceability between the SCR specification and any other (e.g., English) requirements document. The Type Dictionary substantially improves the readability of the SCR specifications, since it encourages enumerated types particular to the modeled domain (e.g., pitch, yaw, roll), as well as allowing subsequent type-checking of the specifications. The Mode Class Dictionary lists the modes (states) of each mode class (finite state machine) in the requirements model. The Assertion Dictionary describes invariants on the system's state. Those invariant assertions that are tagged as "enabled" are then checked during the Simulator's runs of the specifications.

SCR supports three types of tables: Mode Transition Tables, Condition Tables, and Event Tables. Appendix A shows examples of these three table types. (These tables have different names in the User Guide, which are noted in parentheses below.) Upon entering the SCR* toolset, the top-level window is a "Table of Contents" that lists the dictionaries and types of tables. Double-clicking on a dictionary opens its window; double-clicking on a table-type brings up a second-level listing of all tables created so far of that type.

A Mode Transition Table (Mode Class Table) describes a finite state machine in the requirements model. Each row of the table contains a source mode, the event that triggers a transition to the next mode, and the destination mode that is thus reached. Each controlled variable or term has either an Event Table or a Condition Table. An Event Table (Controlled Variable Table) specifies for each mode the events that cause the controlled variable and/or term to change its value. A term is a variable internal to the specification used in the description of the relationships between inputs and outputs. Terms also improve the readability of the specifications by serving as shorthand descriptions. An example of a term is term_OVERPRESSURE_IN_PLENUM_TANK in Table A-7 of the appendix. Lastly, a Condition Table (Term Variable Table) is created for each term or controlled variable to describe the conditions (predicates) that cause the variable to take on different values.

The User's Guide (available on-line with permission from the SCR group at NRL) (Kirby) provides a good introduction to the creation and editing of the requirements specifications. The version we downloaded contained 68 pages of description and examples, with placeholders for some sections that were currently incomplete. The User's Guide describes the required syntax. Also recorded there are which fields of each table and dictionary are optional and which are required for subsequent automated checks.

2.3. Requirements Analysis in SCR

The CC Checker (Consistency and Completeness Checker) tools performs automated syntactic and type checking on the specifications. It also checks for unique names, missing tables, missing conditions in a condition table, and cycles in dependencies among the variables. Mode-reachability is offered as another automated check. The tool also checks that the tables assign a unique value to the controlled variable, term, or mode class (i.e., that the conditions or events in a row are disjoint).

The tool is set up to support rapid debugging of specifications. If the CC Checker reports an error, double-clicking on the error brings up the specification table containing the error, with the error highlighted. A "Messages" field often displays additional debugging information that further helps explain the error. The table can thus be edited and rechecked.

2.4. Dependency Graphs in SCR

Once a specification is syntactically correct, a dependency graph can be automatically drawn. The graph shows which data items depend upon the values of other data items. Monitored variables are lined up as nodes on the left-hand side of the page; controlled variables are lined up as nodes on the right-hand side of the page; modes and terms are shown in the middle of the page; and arrows between the nodes display the dependencies. This formatting not only visually demonstrates unintended dependencies and circular dependencies, but also graphically displays any "orphan" or isolated nodes (often indicating missing requirements). A monitored variable not connected to any controlled variable may indicate a missing requirement; a controlled variable not dependent on any monitored variable may indicate an erroneous specification (Heitmeyer, Kirby, and Labaw). Table A-16 in the appendix shows an example from the DCIU application of a dependency graph.

2.5. Executing SCR specifications

The Simulator tool displays in tabular form changes to the values of the monitored variables, controlled variables, terms, and mode classes as events occur. (See the Appendix for examples). Options to run, step, backup, and restart the simulation exist. A scenario, which is a sequence of pairs of monitored variable names and values, can be stored as a file and retrieved for importation into the Simulator, according to the User's Guide.

Changes to values of the items are highlighted, making it easy to compare the specification's behavior with the intended behavior of the system. Double-clicking on a highlighted (i.e., changed) value will display the associated table that prompted the change. The Assertion Dictionary, which lists properties that can be tested in the Simulator, is linked to the Simulator, so that violation of a (user-enabled) invariant assertion prompts an error message. Again, double-clicking on the error brings up the Assertion Dictionary in another window, with the relevant assertion highlighted. We found the capability for executable specifications with assertion checking to be one of the most useful features of the toolset.

2.6. Maintaining requirements

Changes to requirements can be made either by editing the SCR tables and dictionaries with the SCR editor, or by editing the ASCII specifications in whatever editor the user chooses. This flexibility was useful in one case when a remote network slowed updates to the SCR tables. The user chooses which checks to perform on which tables and dictionaries, by either choosing "All Checks" or highlighting the checks to perform, and by highlighting the tables and dictionaries to be checked (the default is all of them). This allows quick testing of only the tables that have changed.

2.7. SCR support for safety analysis

The assertions in the Assertion Dictionary are invariants that must hold in every reachable state, or invariants that must hold in every pair of adjacent reachable states.

A useful feature of SCR is that it will accept some errors (such as nondeterminism or missing tables) but will flag them, allowing rapid development of an initial prototype, partial specification (e.g., of areas of concern), and evolutionary development of a requirements model. Safety concerns can thus be incorporated into the model at an early stage of development and refined as details are added.

One of the advantages of SCR is that SCR specifications can be imported into automatic theorem provers or model checkers with minimal (although, at this point, manual) translation. This opens the door to the proof of safety properties and liveness properties beyond the invariant checking offered in the Simulator. Owre, Rushby, and Shankar describe how PVS can be used on SCR specifications. Easterbrook & Callahan describe how SCR specifications can be used to drive the model checker, SPIN, providing a counterexample when an invariant fails to hold. Bharadwaj and Heitmeyer describe changes to allow SCR to verify specifications, based on SPIN.

3. Applications to Deep-Space 1 Spacecraft Requirements and Design

This section describes the application of the requirements toolset SCR* (Software Cost Reduction) from the Naval Research Laboratory (NRL) to components of the Deep Space-1 spacecraft at Jet Propulsion Laboratory. Two applications were performed to explore the SCR* toolset's capabilities and to evaluate its usefulness for requirements/design specification and analysis of Deep Space-1 (DS-1). DS-1 is the first spacecraft in the New Millennium Program, a sequence of projects that demonstrates promising new technologies for the space program. In the software domain, this includes innovative software architectures, highly autonomous remote agents, rapid development via a spiral process model, and high-reliability software. In DS-1 Project Safety Plan (D-13533), safety-critical software is defined as software that can command a hazardous function to happen or prevents a hazard from occurring (failure of the software can allow the hazard to happen). Clearly, DS-1’s fault protection software is safety-critical.

3.1. DCIU Fault Protection

SCR was applied to a subset of the functional requirements for the fault protection that the Digital Control Interface Unit provides for the Xenon Feed System of the Deep Space-1 spacecraft. The Xenon Feed System is part of the Ion Propulsion System (IPS), one of the set of devices required to perform Solar Electric Propulsion cruise. The IPS is commanded by the Remote Agent software onboard the spacecraft.

The DCIU fault protection was chosen because it involved a safety-critical component of the spacecraft; because it involved a control structure readily modeled with finite state machines; and because the project had expressed interest in a more thorough requirements analysis of this critical component's behavior.

The SCR specification of the DCIU fault protection produced as a result of our effort is presented in the appendix and described below. This section of the report briefly presents the context for the problem and then describes the SCR specification and automated analysis in some detail. No prior familiarity with SCR on the part of the reader is assumed.

There are two tanks in the Xenon Feed System: the Main plenum tank and the Cathode plenum tank. Each tank has three pressure transducers upstream of it: PA1, PA2, and PA3 for the Main plenum tank and PA4, PA5, and PA6 for the Cathode plenum tank. There are two regulators, each involving two solenoid valves upstream of the transducers. Regulator R1, Solenoid Valve SV1, and Solenoid Valve SV2 control the flow to the Main plenum tank; Regulator 2, Solenoid Valve SV3, and Solenoid Valve SV4 control the flow to the Cathode plenum tank. There are three latch valves involved in the fault protection for the Xenon Feed System: Latch Valve 3 (LV3), upstream of R1; Latch Valve 4 (LV4), upstream of R2; and Latch Valve 5 (LV5), which connects the two flows.

3.1.1. SCR Requirements Modeling

The following subset of the fault protection requirements for the Xenon Feed System, excerpted from the draft version available, (NSTAR Software Requirements Document) were modeled, specified and analyzed.

1. Fault Protection
a) Autonomous Fault Protection
The DCIU (Digital Control Interface Unit) shall autonomously detect and respond to the following faults within the XFS (Xenon Feed System):

> 1. Internal leakage or failure of LV3 [Latch Valve 3] to operate.
>
> Indication: Pressure in plenum tank exceeds nominal by 0.5 psi
>
> Response:
> 1. Close LV3
> 2. Open LV5
> 3. Operate with regulator R2.
>
> 2. Internal leakage or failure of LV4 [Latch Valve 4] to operate.
>
> Indication: Pressure in plenum tank below nominal by 0.2 psi
>
> Response:
> 1. Close LV4
> 2. Open LV5
> 3. Operate with regulator R1.

Besides the obvious simplification of only modeling a subset of the fault protection requirements in this work, several other simplifications were made in developing the requirements model and SCR specifications of the DCIU. For example, the actual voting algorithm for the transducers was not modeled since it involved design details not essential for the requirements and its description was ambiguous in places. Instead, "placeholder" voting logic was used (taking the average of the three transducer values). However, a negative effect of this modeling decision is that simulation of the requirements is then strikingly unrealistic in some scenarios. For example, one "bad" transducer value can thus trigger the declaration of a fault condition even before the other two transducer values are considered. The decision not to include design details thus reduced the value of the simulation for understanding the required behavior of the fault protection.

Internal inconsistencies in the requirements document were noted and passed to the development team. Sometimes modeling decisions were made pending resolution of the ambiguity, and noted in the SCR "Comments" fields. For example, fault protection against internal leakage of LV3 and LV4 is required at one point in the document, but excluded from a list of fault protection elsewhere in the document.

3.1.2. Specification of DCIU requirements

The SCR specifications of the DCIU requirements are in the appendix. Tables A-1 through A-11 present the specifications in the tabular format that SCR* uses. Each latch valve is modeled as a separate finite state machine (mode class). For example, Latch Valve 3 (LV3) is a mode class consisting of three modes: Open, ClosedOK, and LeakingOrFailed.

One of the difficulties that arose in using SCR to specify spacecraft components is how to model commands from one subsystem to another where each subsystem is, at some level, being modeled as a separate black-box. Often what appears to be a controlled variable in one subsystem (e.g., a command from the Navigation Subsystem to the Camera Subsystem to "reset camera") will for another subsystem be a monitored variable whose arrival or value will trigger a mode change. This happens, for example, if a single black-box system is decomposed into several lower-level black-box components. The question of what is an externally visible output may then sometimes be less clear on such a model of an autonomous spacecraft than on an avionics application where there is a pilot to see the output or a monitor on which to display the output. In the literature to date on SCR, there are few examples of how such composition of subsystems should occur.

Moreover, since most of the published examples of SCR usage involve a single mode class, little guidance was available in the literature for describing interactions among mode classes or transitions due to internal events. Workarounds in SCR sometimes detracted from the specifications readability.

In a finite state machine model, transitions between states can be labeled with both the event that trigger the transition and with the output that occurs as a result of the transition. For example, a camera moves from state "taking-picture" to state "ready" at the event "picture completed," with an output message to Navigation, "process picture." In SCR any such output command is modeled separately in another part of the specification as a controlled variable.

Even when the requirements themselves are clear, the process of modeling the requirements in SCR involves experimentation and consideration of alternatives and tradeoffs. As an example, one of NRL's recent papers (Bhaaradwaj and Heitmeyer) describes a system that can be in the calibrated air speed mode at the same time that it is in another mode. That is, the system can be in two modes (e.g., calibrated air speed with the altitude engage mode armed) at once in some cases. Their modeling decision, which preserves both accuracy and the specification's readability, was to represent the calibrated air speed as a boolean term rather than as a mode.

3.1.3. Automated analysis of DCIU requirements

The SCR* toolset can detect some requirements errors automatically via the various analyses that can be performed on the SCR specifications.

Most of the errors found were typographic or syntactical, such as a range having been specified as "[0.400]" rather than "[0,400]." One error involved the failure to consider all possible values of an enumerated term, so the table was incomplete. An example of a coverage error found by the CC Checker was: "ERROR: conditions do not cover all cases for mode sLV3Open and sLV3ClosedOK."

Another error referred to an inconsistent initial state, prompting further examination of the English requirements document. The error message in that case was "ERROR: Assigned value for con_CLOSE_LV3 is Issued but dictionary value is NotIssued." The explicitness of this error message is typical of the user-friendly debugging aids for simple errors in SCR. In this case, the question was whether LV3 is commanded closed if LV3 is Open or ClosedOK and no overpressure has occurred. The automatic CC Checker pointed out the following instance of nondeterminism (same conditions held for the transitions in two cases). For both Solenoid Valve 1 and Solenoid Valve 2, there was no way to distinguish the condition "failure to operate" from "internal leakage" on the basis of the SCR monitored variables and logic. A return to the requirements document confirmed the accuracy of the SCR specifications in not distinguishing the two failures. This in turn led to the more basic requirements question of whether this inability to distinguish a "failure to operate" from an "internal leakage" is intended.

The messages from SCR when an error was detected (e.g., "at or near") and the capability to double-click and bring up a window with the location of the error highlighted sped up debugging of the specifications. The specifications were easy to change and re-check, and encouraged trial of alternative specifications.

3.1.4. Dependency graphs for DCIU

The dependency graph for DCIU is shown in Table A-16. The dependency graph was useful both for displaying unintended relationships among the variables and for cleaning up the specifications by pointing out "extra" or indirect dependencies that could be removed. The dependency graph seemed to help us "see the forest" as well as the trees. The lining up of the monitored variables on the left-hand side of the page and of the controlled variables on the right-hand side of the page gave a useful top-level data flow (or context) diagram. The dependency graph also helped produce more readable specifications with minimal dependencies. It is possible that this could lead to cleaner interfaces in the design, since the graph encourages tinkering to produce a visually pleasing (e.g. reduced number of dependency arcs) product.

3.1.5. Simulations of DCIU requirements

Tables A-13 and A-14 present examples of windows from the simulation of the DCIU requirements. The simulator was useful for exploring out "what-if" questions and disclosed some instances in which the SCR logic used to specify the requirements was incorrect. The process of exercising the SCR specifications in the simulator led to a few surprises. For example, in one case the simulator showed that, contrary to expectations, Latch Valve 5 was not opened in case of an overpressure, although it was being opened in case of an underpressure. This was of value in detecting an inconsistency between the SCR specifications and the requirements document. Upon inspecting the original requirements document, it was found that it correctly specified that LV5 be closed for both an underpressure and an overpressure. However, the SCR specifications did not mirror this requirement. Had the requirements document also been in error (i.e., had Latch Valve 5 not been required to be closed for an overpressure), that would have shown up at this point.

A second example of an error in the SCR specifications detected with the assistance of the simulator was that the simple averaging logic used in the SCR specification to combine the values from the three pressure transducers was inadequate to capture the required behavior of the fault protection. These became clear when entering an extremely large or small value for the first transducer triggered fault protection (erroneously) before all three pressure transducers had been read.

The simulator was also used to check that overpressure values in the pressure transducers PA4, PA5, and PA6 (correctly) don't cause a fault condition when the main plenum tank is being used. In addition, the possibility that the initial conditions (unspecified in the document) for Latch Valve 3 and Latch Valve 4 were wrong in the SCR specifications, was checked via simulation.

3.1.6. Requirements issues

This section lists issues found during the SCR work that involve the DCIU fault protection requirements.

--**Missing requirement**. SCR* requires that an initial condition be specified for each variable. For several variables, it wasn't clear from the English requirements document what the initial value should be. In these cases, assumptions were made about the initial value and documented in the comment field that SCR provides.
--**Inconsistent requirement**. As described above, the English requirements document requires fault protection against internal leakage of LatchValve 3 and LatchValve 4 in one place but doesn't mention those cases in another list of fault protection.
--**Unclear requirement**. There is a requirement to "perform a transducer zero check at the earliest possible opportunity." This was not modeled since it was unclear what was meant.

Martin Feather pointed out two additional issues found through careful reading. These requirements are beyond the scope of the SCR requirements model, but are included here for project convenience of review.
--**No time-out**. When the thruster set-point is being changed, fault protection temporarily uses a wider window between upper and lower trigger points to avoid erroneously declaring that an overpressure or underpressure exists. However, if the process of changing to the new tank pressure is interrupted, the wider window will continue to be used indefinitely.

--**Ambiguous requirement**. The requirement to "use the average of the two closest transducers" in the voting scheme may mean closest to the average or closest to each other.

3.1.7. Safety analysis

Two safety assertions were enabled during execution of SCR's simulator to provide examples of safety properties. These were invariants that specify that DCIU fault protection only occurs when a fault condition exists. The two invariants that the DCIU fault protection must satisfy are: (1) A command to close Latch Valve 3 only occurs when an overpressure condition exists and (2) A command to close Latch Valve 4 only occurs when an underpressure condition exists.

With the assertions in the Assertion Dictionary enabled, the simulator tested that they held in every state reached during the simulation scenarios. No new errors were found, providing some verification that in at least this set of event scenarios, fault protection was not invoked when it shouldn't be.

Assertion checking during simulation is easy to perform and provides some limited assurance that at least these properties hold in the states that the simulator has reached. All the assertions entered in the Assertion Dictionary for the DCIU fault protection were invariants on a state. We anticipate that in some applications use of invariants that assert relationships between old and new values of a variable (e.g., to show that a counter increments or to show termination) would be useful.

3.2. DS-1 Standby Mode Transition of System Fault Protection

3.2.1. Overview

The DS-1's autonomous, on-board fault protection is the onboard flight software measures taken to reduce the likelihood of a single fault that will lead to a total or partial loss of spacecraft functionality or mission-critical data [DS-1 Project Policies and Requirements]. The DS-1 system fault protection consists of monitors, event reporting services, and fault protection response tasks. These three components work together to notify the ground operations on Earth in the event of a fault, and to determine and execute the appropriate fault response(s). For persistent faults, the fault protection's response will lead the spacecraft to an appropriate Standby Mode, which will then put the spacecraft in a power positive and communicative state with Earth to allow ground operations to solve the problem on the spacecraft.

The DS-1 system fault protection is designed to liberally use the Standby Modes. The design philosophy is to get the spacecraft into a safe state when necessary, rather than going through exhaustive measures to diagnose problems and continue sequence execution. There are three Standby Modes: EARTH_STANDBY, SUN_STANDBY_SRU, SUN_STANDBY_SSA. Depending on the health status of the spacecraft hardware components, appropriate Standby Mode is chosen by the Standby Mode Transition Logic of the DS-1 system fault protection.

In the EARTH_STANDBY mode, the spacecraft +x-body axis is pointed at the earth using the spacecraft centered inertial reference obtained by the Stellar Reference Unit (SRU) and the knowledge of Earth body. There are two planned EARTH_STANDBYs: 1. At Deep Space Network (DSN) acquisition following a successful launch sequence, and 2. During scheduled DSN communication windows. EARTH_STANDBY requires working Sun Sensor Assembly (SSA), Stellar Reference Unit (SRU) and Inertial Measurement Unit (IMU). This mode is used only in the absence of any hardware and power failure.

In the SUN_STANDBY_SSA mode, the Sun Sensor Assembly (SSA) is used to measure the sun position. This mode is appropriate for power related faults, because the solar arrays will be pointed at the measured sun vector. SUN_STANDBY_SSA is the Standby Mode of choice during spacecraft anomalies.

In the SUN_STANDBY_SRU mode, the stellar reference unit (SRU) is used to determine the inertial attitude of the spacecraft to align the spacecraft with the sun vector. It is the only Standby Mode that does not use SSA and IMU.

Depending on the state of the spacecraft health, the on-board Standby Mode logic will put the spacecraft in the pre-defined Standby Mode when needed. It is important that the correct Standby Mode be executed to prevent further degradation of the spacecraft's health and to facilitate ground operation's communication with the spacecraft and their attempts to resolve problems in the event of a critical anomaly.

3.2.2. Approach

This study takes an in-depth look at the spacecraft's onboard Standby Mode Transition Logic from the normal state (FP_NORMAL) to three Standby Modes. The approach taken in this study was to capture the requirements (or rules) in the SCR Assertion Dictionary, translate the Standby Mode design logic into SCR specifications (tables), and then validate the design logic against requirements/rules using the SCR Simulator. Two versions of the SCR models were created, the first one was based on the DS-1 Fault Protection Notebook draft (3/97 version) and the Standby Mode Entry Logic diagram dated 4/21/97, the second model was constructed based on the updated the DS-1 Fault Protection Notebook (Version 1.0).

A system black box for the Standby Mode Transition was constructed (see Figure 1). The system black box provides a visual tool to facilitate the understanding of the problem domain. This black box is not a closed loop, feedback system, and the diagram reflects that. This study models the Standby Mode Transition black box's behavior and the resulting spacecraft Standby Mode configuration.

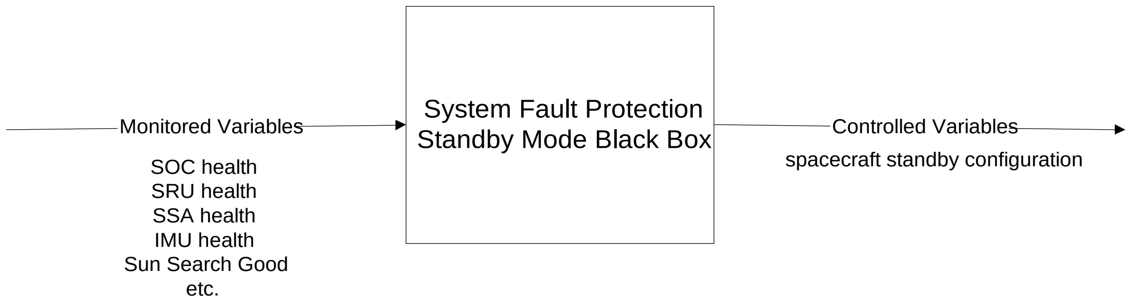

Figure 1.Standby Mode Transition Black Box

The next steps were to identify SCR specification goals and to construct the SCR tables. The specification goals were the requirements/rules taken from the DS-1 Fault Protection Notebook. These goals were later entered in the SCR Assertion Dictionary to be checked against the design logic using the SCR Simulator. Creating SCR tables was not a straight forward effort. By trial and error, it was found easier to first describe the mode transition and configuration change scenarios taken from the design logic diagram. This information was captured in a transition/configuration table (see Table B-1 in the appendix).

With the system black box diagram and the transition/configuration table in place, the key SCR elements fell into place: monitored, controlled, and term variables; mode transition, event , and condition tables (see *SCR* Toolset: The User Guide* for definitions of these terms). The use of transition/configuration table facilitated the construction of SCR tables. The transition/configuration table was similar to the SCR Mode Transition Table in format. Items in the transition/configuration table became elements in the Model Transition, Event or Condition Tables. The SCR dictionaries and tables created for this study can be found in the Appendix B.

3.2.3. Discussion

3.2.3.1. Problem Domain

The problem under study was an open loop system containing hardware and software components. The SCR Simulator was able to perform assertion checks for this environment. For example, using Assertion Dictionary to check against the design logic, the Simulator was able to show for the draft version model that when the battery state of charge (mSOC) changed from normal to emergency, the spacecraft standby mode configuration (cEndConfig) did not change in value, leading to violation of an assertion: When mSOC is Power Alert or Power Emergency, the cEndConfig should be in either SUN_STANDBY_SSA (SunSSA) or SUN_STANDBY_SRU (SunSRU). This occurred because the Standby Mode Entry Logic diagram dated 4/21/97 showed only the Power Alert entry into the Standby Mode Entry and didn't include the Power

Emergency case. This design logic had gone through several iterations of update, and this problem did not appear in the Version 1 model.

When specifying mode transitions and events for the Standby Mode Transition, two approaches were tried. One way was to model the system response in a sequential matter, as one event leads to another according to the design logic (the sequential model). The other way was to rely heavier on the system condition leading to transitions and events (the condition model). It was found that the condition model was a better approach for this simulation study, because the interest was in checking the end configuration based on the condition of the system (and not on sequence of events). The sequential method cannot model this system response as well.

3.2.3.2. SCR Usage

There are various ways to utilize SCR's capabilities. One way that has been demonstrated in this case study is to validate a design against requirements (or rules). The strategy and techniques used in this study have been described earlier and are summarized in Table 1.

Table 1. Strategy, Techniques and Procedure Used

Strategy	• Capture requirements/rules in the SCR Assertion Dictionary • Specify design logic in SCR tables (mode transition, event and condition tables) • Validate design against requirements using SCR Simulator
Techniques	Construct the following items to help identify SCR elements • System black box • Transition/configuration table
Procedure	See sample procedure in Section 4

It was found that during initial designing of the SCR tables, when changes are frequent, the use of a word processor (such as Microsoft Word) rather than SCR will be much easier. Input these tables in the SCR when the tables are pretty much finalized and are ready for the SCR's consistency and completeness checker (CC Checker). This will save time and be much easier, especially to a new SCR user.

Before validations or analysis of the SCR-specified system, SCR statements should be checked for completeness and correctness using the CC Checker. One should also check for the correctness of the assertion statements when such table exists, by running test cases using the Simulator. It is possible to find both the modeling errors and the design/requirements errors during this step.

The CC Checker is a very useful tool. It provides a variety of checks, such as disjointness, coverage, consistent initial state, and mode reachability checks. The *SCR* Toolset: The User Guide* (Kirby) provides detailed usage information. Warnings are provided by the CC Checker when potential errors exist. It is possible that certain warnings are acceptable for the particular problem domain. Both models created for this case study had acceptable coverage and disjointness warnings. Other warnings led to the correction and refinement of the models.

It is important to do the SCR tables as the requirements/design say; don't try to clarify, interpret, or correct the requirement/design (unless noted). This avoids the situation where the SCR specifications are correct, but correction to the error source (requirements/design) is neglected.

3.3. SCR issues

SCR* is a continually evolving tool, so the version of SCR* that was used for the DCIU requirements specification and analysis (downloaded from NRL in February, 1997) differs somewhat from the most current version. For example, the version of SCR* described at the recent NASA Langley Formal Methods Workshop has new capabilities (e.g., an "@CHANGE command) and enhancements (e.g., a link to the SPIN model checker). The following items are thus provided not as a current status report but as a checklist for future users of SCR. It is our hope that we can save them time in their use of SCR by documenting known inconveniences and constraints. It is also our hope that this list will be useful to SCR developers as they develop future versions of SCR.

Note that some of these items are almost certainly the result of our misuse or limited experience with the toolset. The reader should remember that any verification of software tends to yield some items due to "analyst error," and take our comments as a snapshot of our use of a particular, now somewhat outdated version of SCR. The SCR version used for the case studies were scrtoolA.solaris dated October 1996 and the modified HP version, downloaded in February 1997. [1]

1. Difficulties

    1.1. Takes time for an user to become proficient in SCR

    1.2. Need to validate the SCR model before verifying the requirements/design

    1.3. Latest User Guide is inconsistent with the latest SCR version (e.g. use of INMODE, incrementing term variable, etc.)

        1.3.1. A prime (') following a monitored variable refers to the variable's value in the new mode, according to the User's Guide's syntax, but in the tool the prime preceding the variable refers to the variable's value in the old mode, whereas the variable with no prime refers to the variable’s value in the new mode.

        1.3.2. ALWAYS is an "undefined symbol" for the condition table, according to the error message, but NEVER is accepted in some versions.

    1.4. Assertion statement syntax unclear in User Guide; does not take “ < = >” (If-and-only-if) expression syntax in the assertion table

    1.5. The User's Guide still contains several blank sections; naming conventions are in some places inconsistent with the tool (e.g., the naming of the types of tables); and Table 9, not Table 6 as stated, describes the syntax of the expressions.

    1.6. Detailed information early in the User's Guide about how to accomplish the printing of files would be useful. In particular, this requires Framemaker or a tool that prints the contents of a window, such as UNIX's xgrab or freeware that handles MIF files.

    1.7. There is no quick way to rearrange the rows in a table for readability (e.g., to group all the monitored variables together).

    1.8. The "All Checks" option in the CC Checker succeeded even when applying the checks one-by-one yielded errors. This masked the errors in the specification.

    1.9. Closing any window in SCR with the icon rather than the pull-down menu aborts the tool--and erases any changes to this or other tables that have not been saved.

2. Limited capabilities

    2.1. Current version does not take @T(Inmode) statement as described in the user guide. For transition table, this would address the system response at the time of entry to a specified new mode under defined condition. Without this option, the capability to better describe the system is limited.

    2.2. Limited mathematical functions/operators for expression (e.g., the MOD function or the modulus operator is not available)

    2.3. Cannot do timed-simulations (e.g., the condition “ POR_counter will reset to zero 1 hour after the last POR” can only be simulated using a monitored variable to manually indicate that 1 hour has elapsed.)

[1] Another version was also tried (scrtoolA.solaris5.5.1, dated Feb. 1997), but it had a problem with Assertion Table. The Simulator reported violations of all the assertion statements marked enabled whenever an event was executed, even if the executed event did not relate to the assertion statements.

2.4. Cannot depict dependency among monitored variables (e.g., if mSSA_Good = FALSE then mSunSeenGood = FALSE, where both are monitored variables)

2.5. Restrictive cycle dependency: The tool prevents variable cycle dependency, which in spirit is good, but prevents the description of the real system. For example, the following term variables cannot be specified: The new tPOR_count depends on tRebootStatus which in turn depends on the existing tPOR_count.

3. State machine simulation. SCR focuses on system's state transitions, therefore, a real simulation cannot be performed (e.g. increment_counter only increments when there is a change in the state value, not at each trigger of a stimulus)

4. Sample Procedure for SCR Usage

The procedure which summarizes the steps taken in performing the SCR case studies is provided in this section. The intent for this sample procedure is to provide a procedural guideline to new SCR users in specifying requirements and/or design logic using SCR. This black box approach to SCR modeling is an iterative process.

1. Getting acquainted with SCR

Sources of information for SCR include the *SCR* Toolset: The User Guide* (Kirby) and various published papers (e.g., Heitmeyer, Bull, Gasarch, Labaw)

2. Understanding and Defining the Problem Domain

2.1. Become familiar with the problem domain
Example: Understanding the purpose and function of the DS-1 System Fault Protection and its Standby Transition Logic (SFP)

2.2. Define the problem statement
Example: Modeling DS-1 SFP Standby Mode Transition Logic and validate the model against the model goals identified. Note: The objective is to model and analyze a system behavior (or subset of a system).

2.3. Identify system mode class(es) to be modeled
Example : System Fault Protection Standby Mode

2.4. Construct the system black box (enclosing the system being modeled).
Example: Figure 1.

2.4.1. Define the system boundary. Example: SFP system, including H/W and S/W

2.4.2. Identify stimuli (inputs) to the system (these are the monitored variables).

2.4.3. Identify external system responses (these will likely be the controlled variables). They are the system's reactions to the external stimuli or system's internal variables and are visible to the outside of the system black box. These responses are the interest of the problem.

2.4.4. Identify internal system responses (these may be the term variables). They represent the internal system status which have direct affect on system's response to the stimulus, but are invisible outside the system.

2.5. Identify goals, initial modes, and assumptions of this modeling/analysis effort.

3. Designing SCR Tables

3.1. Construct the mode transition and system configuration change (response) table. A transition/configuration table is similar to the transition table, but it also includes system events and conditions as well (see Table B-1). Through iterations of this transition/configuration table, variables (monitor, term, control) will easily fall into place. Examples of information sources include Requirements document, design document, and design logic diagram. For the Standby Transition Case Study, we used the design logic diagram as information source.

3.1.1. Define mode class, modes, terms, and condition variables. These may go through frequent changes in the beginning.

3.1.2. Review and verify

3.1.2.1. check for correctness, e.g., check the transition/configuration table against the logic/design diagram

3.1.2.2. determine/re-evaluate variables for monitor/controlled/term classifications

3.2. Identify rows in the transition/configuration table that are mode transitioning or events, and those that are conditions. Re-organize the table as needed.

3.3. Construct SCR tables (e.g. variable tables, Event Tables, Condition Tables, Mode Transition Table, etc.)

- It may be easier to first define SCR tables using a word processor such as Microsoft Word, then generate the tables in SCR when the tables are finalized.
- Mode Transition Tables and Event Tables contain @T and @F statements.
- Condition Tables do not contain @T, @F statements

4. Verifying the SCR model for correctness

The main focus here is to verify the correctness of the model, although it is possible to find errors in requirements and design as well.

4.1. Create the SCR tables in SCR

4.2. Run the CC Checker and make corrections.

- It is possible to find requirements/design errors here
- The reported errors are warnings. Review the warnings and make appropriate corrections. It is possible that some warnings may be acceptable.

4.3. Check the Dependency Graph. There should be no cycles in the New Dependency Graph. An example of a new dependency would be "Value1 in the current state depends on Value2 in the current state". The acceptable dependency is "Value1 in the current state depends on Value2 in the *previous* state". The SCR Simulator does not allow new dependencies. The Simulator in the SCR version used would abort if there is a new dependency (cyclic problem).

4.4. If Assertion Table is used, run test cases to check each assertion statement for correctness.

4.5. Run through various scenarios to check the SCR model for correctness.

5. Performing system validations

5.1. Run the SCR simulator, covering critical scenarios at a minimum. It is best to hook simulator up with an automated checker to automatically run through all possible conditions the system can be in.

5. Conclusion and Recommendations

SCR displays many of the features that appear likely to be key in enhancing the quality of requirements and design, and in supporting their specification and analysis in the future. These features include specifications readable by non-experts, executable specifications, seamless links between the specifications and tools to check these specifications, user-friendly graphics and editing capabilities (multiple windows, highlighting of errors, debugging messages), syntax and type checking, and visualization of dependencies in graphical format. SCR's support for changing requirements includes easily-updatable tabular notations and flexible checking of the consequences of changes.

Since the specifications are stored in a special ASCII format (SSL; see User's Guide for details), they are readily transportable across platforms. This feature was especially valuable to us since we were emailing updates back and forth and running different versions of SCR on different platforms (HPUX and Solaris). A larger project could readily have shared the SCR specifications via web postings.

In a few cases, front-end mockup displays have been tied to the SCR requirements model so that customers of the system can animate or execute the requirements. This has proven valuable in requirements elicitation, strengthening the developer's assurance that the model's behavior matches the customer's view of the system requirements. In addition, the most recent version of SCR* incorporates model-checking (Bharadwaj and Heitmeyer) and work is underway to strengthen the link between SCR* and PVS (Owre, Rushby, and Shankar; Archer and Heitmeyer). Incorporation of a testcase-generation tool, is also being pursued by NRL.

The SCR case studies have shown that a requirements/design analysis tool such as SCR has practical applications in safety critical systems. Our studies showed that such a tool provides helpful automated checks such as disjointness, coverage, consistent initial state, and mode reachability checks. Additionally, the simulator assists in the analysis of the requirements and/or design. The studies also showed that the SCR technique can be applied to system as well as software components. Although not demonstrated in these studies, the effectiveness of SCR as a validation tool can be amplified when it is coupled with an automated testcase generator, to automatically check out all possible scenarios.

There is a learning curve associated with the use of a requirements analysis tool such as SCR. It takes time for a SCR user to become proficient in constructing SCR models. The construction of a SCR model also takes some time. Once constructed, the SCR model needs to be checked for correctness before starting a full-blown simulation. Afterall, construction of an SCR model is like writing a computer program. Computer programming skill takes time to master, and a computer program requires testing before application use.

Many of the uses to which SCR has been put do not fit the use for which it is intended. For example, many uses of it have been for reverse engineering, although it is intended to be a tool for initial project development. (It is worth noting, however, that the original application of SCR was for the reengineering of the A-7E project). SCR is intended to provide a black-box requirements model of the system's interaction with its environment, but we and other users have included a significant amount of internal state information in our specifications as well as some design information. SCR is intended for use in a development environment, but we and others have used it mostly for verification and validation of requirements and design.

In part these mismatches are due to the ease with which SCR's flexibility lends itself to misuse. The justification is that the task must be completed with whatever tool is available, and SCR's strengths recommend it for many tasks. A consequence of the mismatch is that it is difficult to judge to what extent the problems that we have experienced are because we applied the tool to a task it was not intended to solve. Thus, while we can clearly state SCR*'s successes, we are still unclear as to what are its limitations. It is predictable that some systems will not lend themselves to the 4-variable model nor to a finite state machine description, although the applications described here did.

SCR can support safety analysis in several important ways. First, any method such as SCR which allows a model to be precisely stated and checked for consistency, omitted cases, and cyclic dependencies will promote safe systems. In particular, SCR's capability to run simulations of the specifications and to test that key invariants are preserved in those scenarios assists the proof of safety properties. Secondly, being able to link the SCR specifications to more sophisticated formal methods such as theorem provers and model checkers further enhances SCR's appropriateness for critical applications. Thirdly, safety-related errors often occur at the interfaces of hardware and software. The 4-variable model underlying SCR promotes a perspective in which the distinction between what is externally visible and what is visible by the software is explicit. Fourthly, safety-related errors often occur when the requirements as understood and implemented fail to match the actual requirements needed for the system to function correctly in its environment. SCR specifications are easy to read, promoting review by both experts and clients, and can be linked with a user-friendly visual representation of the behavior.

We do not at this point in time recommend SCR for a production environment. As of now, use of SCR has been largely limited to research groups at companies, NASA, and universities. This is likely to change as larger problems are tackled with SCR* and reported in the open literature; as the toolset stabilizes; and as more documentation and training become available (at this point NRL is the major source of both). The infrastructure for SCR* usage has matured rapidly in the last year, but adoption by a project for software development at JPL at this time is premature.

Acknowledgments: We thank Sun Matsumoto, George T. Chen, Sima S. Lisman, Paul H. Davis, Martin Feather, and John Kelly at JPL; Steve Easterbrook at the NASA IV&V Facility; Connie Heitmeyer, Bruce Labaw and James Kirby at NRL; Steve Miller at Rockwell-Collins; and the Cornell University Computer Science Department for their help.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial or noncommercial product, process, or service by name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, or the California Institute of Technology.

6. References

Archer, M. M. and C. Heitmeyer, "Human-Style Theorem Proving Using PVS," TPHOLs 97, Aug, 1997.

Bharadwaj, R. and C. Heitmeyer, "Applying the SCR Requirements Method to a Simple Autopilot, Proceedings of the Fourth NASA Langley Formal Methods Workshop, September, 1997.

DS-1 Fault Protection Design Notebook, JPL, Draft Version, March, 1997.

DS-1 Fault Protection Design Notebook, JPL, Version 1, July 30, 1997.

DS-1 Project Policies and Requirements, DS1 6446-210, JPL D-13518, Version 2, August 30, 1996.

DS-1 Project Safety Plan, PD-6446-207, JPL D-13533, October, 1996.

DS-1 Standby Mode Entry Logic, JPL, 4/21/97 version.

Easterbrook, S.M. and J. R. Callahan, "Formal Methods for Verification and Validation of Partial Specifications: A Case Study," Journal of Systems and Software, 1997.

Heitmeyer, C., A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," Proceedings of COMPASS 95, June, 1995.

Heitmeyer, C., J. Kirby, and B. Labaw, "Tools for Formal Specification, Verification, and Validation of Requirements," Proceedings of COMPASS 97, June, 1997.

Lutz, R. "Targeting Safety-Related Errors During Software Requirements Analysis, The Journal of Systems and Software, vol. 34, Sept, 1996.

Miller, S. and K. F. Hoech, "Specifying the Mode Logic of a Flight Guidance System in CoRE," Rockwell Collins Avionics Report, August, 1997.

NSTAR (NASA Solar Electric Propulsion Technology Applications Readiness) Flight System Thruster Element Technical Requirements Document, ND-310, D-13638, Version 4, August 6, 1996.

Owre, S., J. Rushby, and N. Shankar, "Analyzing Tabular and State-Transition Specifications in PVS," SRI CSL Tech Report, June, 1995.

RE'97. Third IEEE International Symposium on Requirements Engineering, Annapolis, MD, Jan 6-19, 1997.

Kirby, J., SCR* Toolset: The User Guide, January 21, 1997.

Appendix A. SCR Models and Analyses: DCIU Fault Protection

List of Tables:

Appendix B. SCR Models and Analyses: Standby Mode Transition

List of Tables: